Dynamical Systems: Chaos, Numerical Analysis, and Complexity

January 28, 2026

Part 1

1.

$$\frac{dx_i}{dt} = x_i(1 - x_i - \gamma(x_{i-2} + x_{i+1})), \quad i = 2, 3 \ldots, n-2,$$

$$\frac{dx_0}{dt} = x_0(1 - x_0 - \gamma(x_{n-2} + x_1) + \mu x_{n-3}),$$

$$\frac{dx_1}{dt} = x_1(1 - x_1 - \gamma(x_{n-1} + x_2)),$$

$$\frac{dx_{n-1}}{dt} = x_{n-1}(1 - x_{n-1} - \gamma(x_{n-3} + x_0)).$$

a) To find a non-trivial equilibrium solution we require the RHS of the system to equal 0. If we set each term in the RHS excluding the x_i pre-factor to be 0 then have a linear system we can solve using np.linalg.solve. Excluding the pre-factors ensures the solution is non-trivial.

To find the initial perturbation vector $\tilde{\mathbf{x}}_0$ that maximises the energy ratio $e(t = T)/e(t = 0)$ we can linearise the system of ODEs. If our original system is $\frac{d\mathbf{x}}{dt} = f(\mathbf{x})$ then a 1st order Taylor expansion yields

$$\frac{d\tilde{\mathbf{x}}}{dt} \approx \mathbf{Df}(\overline{\mathbf{x}}) \cdot \tilde{\mathbf{x}}$$

Then we can use the solution to the maximum growth problem as discussed in lectures. If we define the matrix exponential $\mathbf{A}(t) = \exp(\mathbf{Df}(\overline{\mathbf{x}}) \cdot t)$ then the initial condition which produces the maximum growth at time T is the leading eigenvector of $\mathbf{A}(T)^T\mathbf{A}(T)$ with maximum growth equal to the leading eigenvalue $\frac{|\tilde{\mathbf{x}}(T)|^2}{|\tilde{\mathbf{x}}_0|^2} = \lambda_1$. This is equal to the energy ratio by definition of the dot product. We calculate these using np.linalg.eig.

b) part1q1a provides a theoretical maximum energy ratio at time $T = 50$ based on dynamics from a linear approximation close to an equilibrium point. In contrast, part1q1b uses simulate1 to capture the non-linearity in the system and tracks the true energy evolution, up to errors from scipy.integrate.solve_ivp.

part1q1b returns the difference between the energy ratios relative to the energy ratio calculated using simulate1 in part1q1b (since simulate1 tracks the true energy evolution of the system). This number is equal to 0.000238 or 0.0238%. Since the relative error is extremely small we see that our results are consistent. This indicates that the linear approximation we used in part1q1a is indeed a good approximation to the original system.

c) part1q1c runs the function part1q1a multiple times, varying the time at which the energy ratio is maximised. We generate a graph of the maximum energy ratio (calculated using the linear approximation) against the value of T; see Figure 1. Visually we can see the graph appears exponential, so we plot an exponential fit of the curve using scipy.optimize.curve_fit. This matches the behaviour of the original graph well, and for good measure we also plot a semilogy plot; see Figure 2. Since this is a straight line we can conclude that the relationship between maximum energy ratio and T is exponential.

Using scipy.stats.linregress we find the gradient of this line to be 0.46318 (5 s.f.) and the intercept to be 0.0. So this gives us the relationship

$$\max_{\tilde{\mathbf{x}}_0} \frac{e(t = T)}{e(t = 0)} = \exp(0.46318 \cdot T).$$

In the function part1q1c we also find the eigenvalue of the Jacobian matrix evaluated at $\overline{\mathbf{x}}$ with the largest real part. This value is 0.23159 (5 s.f.) and is almost exactly half of the gradient found earlier.

To justify this we can look at spectral theory. If $\mathbf{A}(t) = \exp(\mathbf{Df}(\overline{\mathbf{x}}) \cdot t)$ then the linearised system has solution $\tilde{\mathbf{x}}(t) = \mathbf{A}(t) \cdot \tilde{\mathbf{x}}(0)$. As such the energy ratio is equal to $\frac{\tilde{\mathbf{x}}(T)^T \tilde{\mathbf{x}}(T)}{\tilde{\mathbf{x}}(0)^T \tilde{\mathbf{x}}(0)} = \frac{\tilde{\mathbf{x}}(T)^T \mathbf{A}(T)^T \mathbf{A}(T) \tilde{\mathbf{x}}(T)}{\tilde{\mathbf{x}}(0)^T \tilde{\mathbf{x}}(0)}$.

From ODE theory we know that, since we are looking close to an equilibrium solution, the growth in the energy ratio will be dominated by the eigenvalue of $\mathbf{A}(T)^T \mathbf{A}(T)$ with the largest real part.

From linear algebra we find the key observation that explains the relationship between these 2 values: if the eigenvalues of $\mathbf{Df}(\overline{\mathbf{x}})$ are $\lambda_1, ..., \lambda_2$ then $\mathbf{A}(T)^T \mathbf{A}(T)$ has eigenvalues $e^{2\lambda_1 t}, ..., e^{2\lambda_n t}$. This is exactly what we have shown computationally, i.e.

$$\frac{\text{exponential parameter}}{\text{real part of dominating eigenvalue}} \approx 2.$$

From 1.b) we know that our linearised system is a good approximation, and so we should expect that the relationship $\exp(0.46318 \cdot T)$ will also be reasonably accurate for the original system.

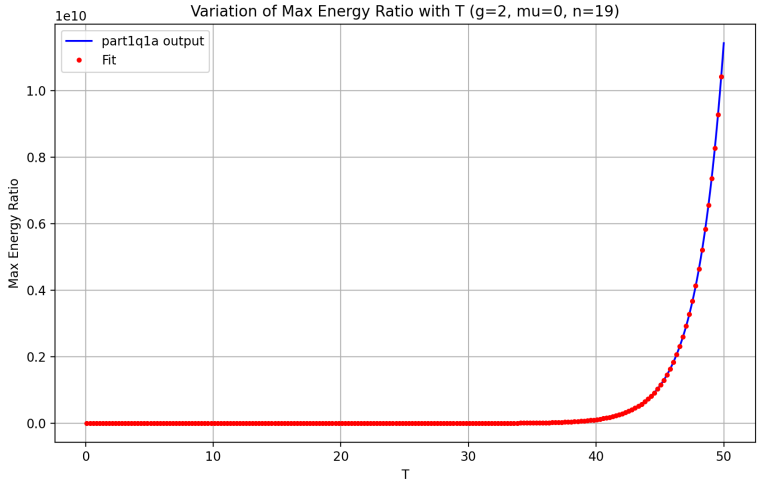

Figure 1: Graph to show how the maximum energy ratio evolves with $0 < T \leq 50$

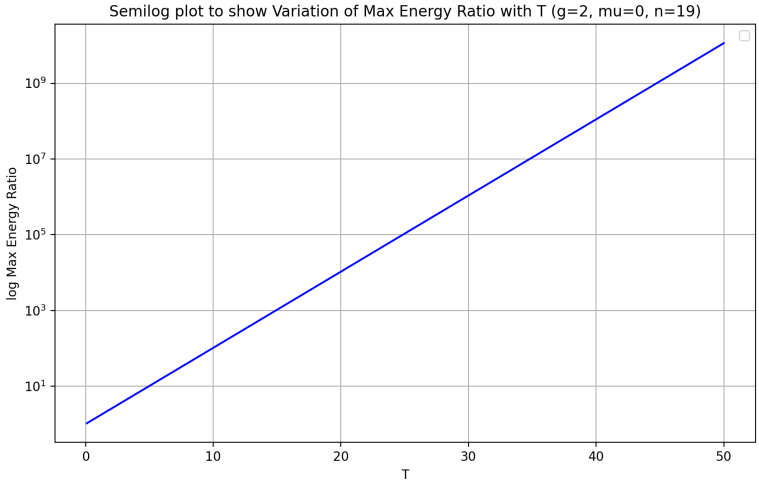

Figure 2: Semilogy plot to show how the maximum energy ratio evolves with $0 < T \leq 50$

2.

n=9:

We can visualise the solution to the system when $n = 9$ by setting flag_display to True. We use the parameters $Nt = 20000$ and $tf = 1000$, which are sufficiently large to observe the long-term dynamics of the system. The resulting contour plot (Figure 3) shows this behaviour (truncated for visualisation).

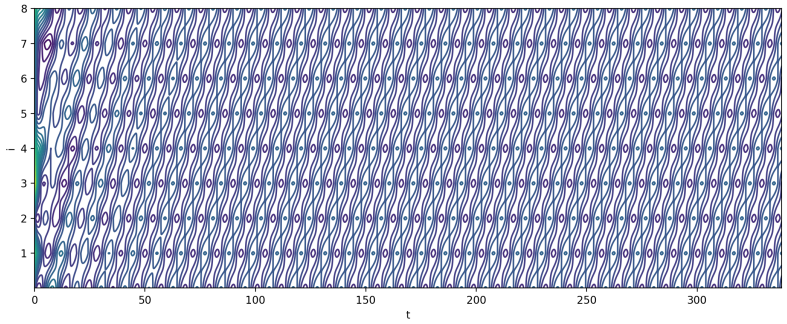

Figure 3: Contour plot for system with parameters $n = 9$, $Nt = 20000$ and $tf = 1000$

We can clearly see the impact of the initial condition on the contour plot. To be completely sure we remove the transient, we will discard data preceding $t = 200$, corresponding to a transient_cutoff parameter value of $\frac{200}{1000} = 0.2$ when calling simulate_and_trim.

We can investigate if we have chaotic dynamics of the system by looking at the Fourier coefficients.

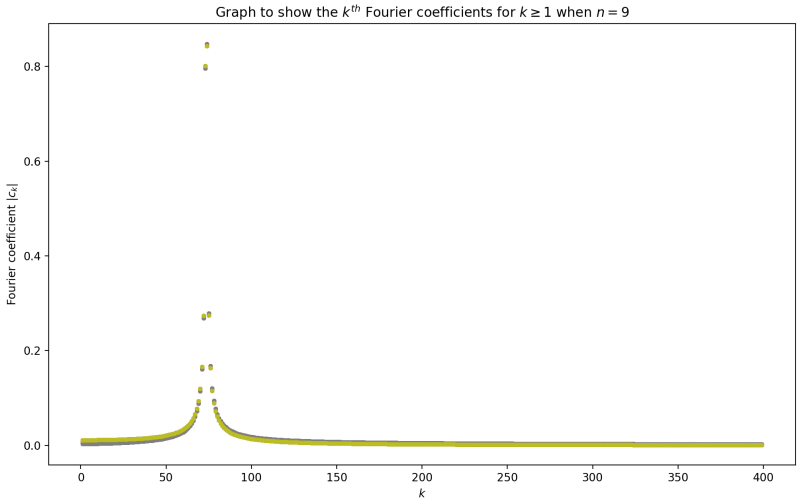

Figure 4: Plot of the first 400 Fourier coefficients for the system when $n = 9$, excluding c_0

Figure 4 illustrates how the Fourier coefficients of the system behave. Since we have a small number of peaks at specific values of k as opposed to a broad and unpredictable spectrum, we have an initial indication that the system does not exhibit chaos for $n = 9$. (Note we exclude c_0 since this corresponds to the mean of the dataset and does not drive the Fourier transform of the system.)

We can now investigate the qualitative dynamics of the system by plotting a time series.

The time series plot shows how we have simple sinusoidal oscillations in all components, showing that the system is periodic. Hence we have no time-dependence; the system's behaviour is determined only by its current state and not the time one observes it at.

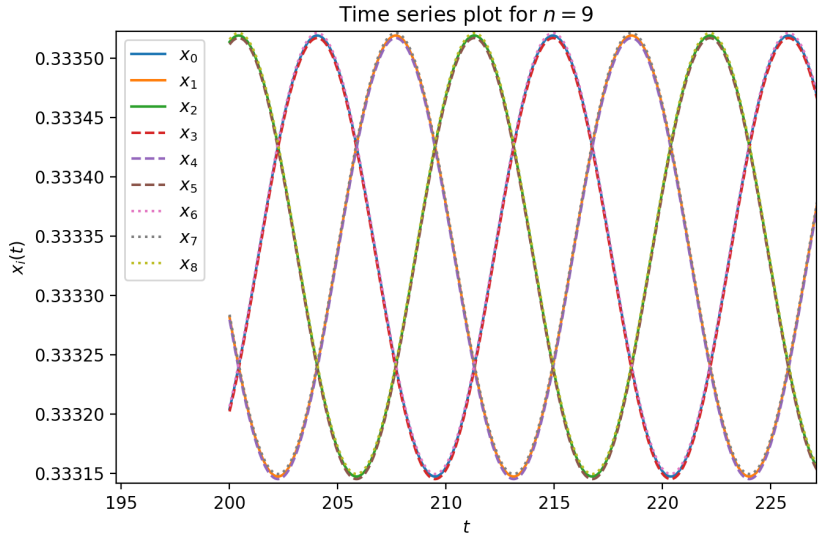

Figure 5: Time series plot when $n = 9$ to show how each component of the system evolves (truncated). Curves are shifted up and down slightly to ensure visibility when they overlap.

From this time series plot we can estimate the frequency of the system's oscillations using scipy. We get $\frac{1}{10.88...} = 0.091891$ Hz to 5 s.f. We can also verify this by looking at the value of k with the greatest Fourier coefficient, which is 73. Using this we estimate the frequency of oscillation to be $\frac{73}{0.8*1000} = 0.09125$ which is extremely close to our original estimate.

From the time series plot we can also see that every 3rd component is synchronised from the overlapping curves (note the different linestyles of curves in Figure 5). Furthermore we see that we have a constant phase difference of approximately one third of the time period between each group of synchronised components; this suggests that our components will always be correlated. The synchronisation/phase difference can also be observed from the contour plot (Figure 3): you can see that along lines of $i = 1, 2, ..$ the patterns repeat with a small shift in the t-axis.

We now find the correlation dimension using the function find_corr_dim, which makes use of scipy.spatial.distance.pdist. Note that we do not need to implement a time delay as described in lectures since the components will always be correlated to some degree.

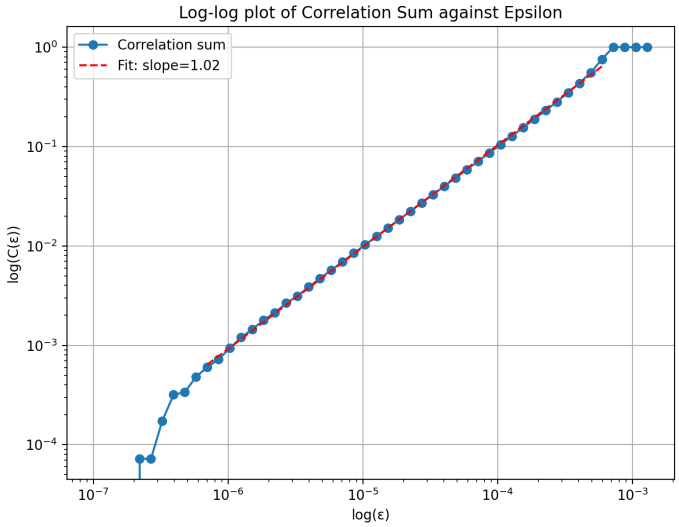

Figure 6: Log-log plot of $C(\varepsilon)$ against ε with a straight line fit for $n = 9$

Here we create a range of epsilon values that includes the entire range of distances, and then we find the correlation sums for each epsilon. Plotting this on a log-log graph and finding the gradient of the linear region gives an estimate for the correlation dimension as described in lectures.

We know from lectures that for an autonomous system of ODEs, a necessary condition for chaos is a correlation dimension greater than 2. Since our correlation dimension is 1.02 we know that for $n = 9$ the system does not have chaotic dynamics.

All of this evidence together strongly suggests non-chaotic dynamics for the system, and that it has sinusoidal oscillations (every 3rd component being synchronised).

n=59:

Here we use the parameters $Nt = 20000$ and $tf = 10000$; a greater final time is required to understand the long-term behaviour of the system. We create the contour plot below:

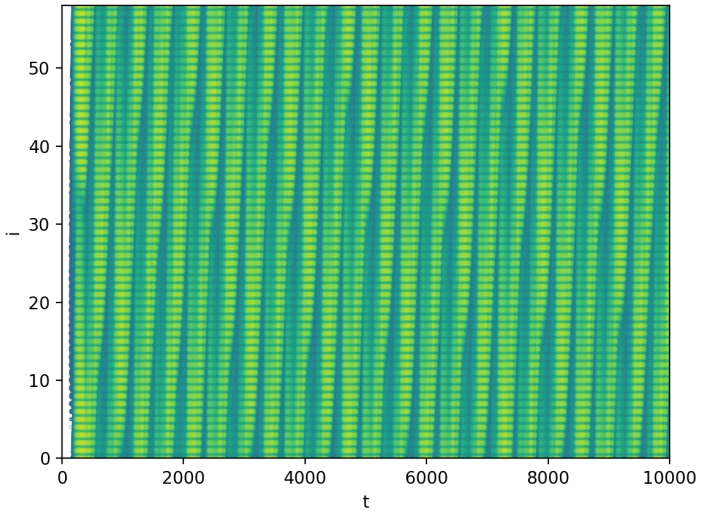

Figure 7: Contour plot for system with parameters $n = 59$, $Nt = 20000$ and $tf = 10000$

It is slightly harder to see the transient behaviour here; we are excessive and use a transient_cutoff value of 0.4 to be careful.

Again we plot a graph of the Fourier coefficients as shown below:

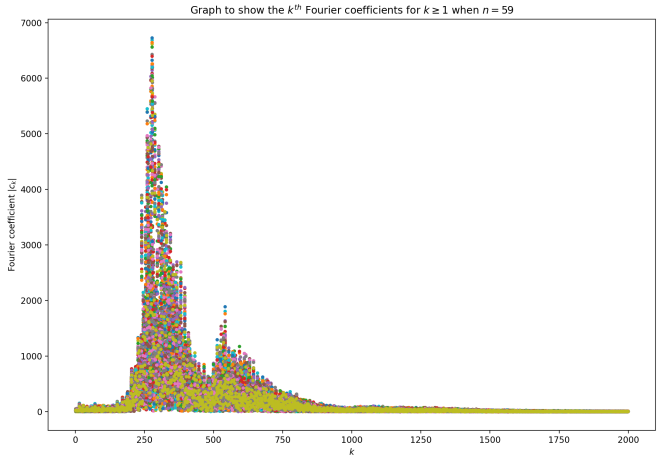

Figure 8: Plot of the first 2000 Fourier coefficients for the system when $n = 59$, excluding c_0

The graph of the Fourier coefficients is in stark contrast to that of $n = 9$; here we see the distribution of the coefficients is much more irregular, unpredictable, and sporadic. This indicates that the system is not driven by a dominant frequency, and that the system is likely to be chaotic. We can investigate this further by computing a correlation dimension again.

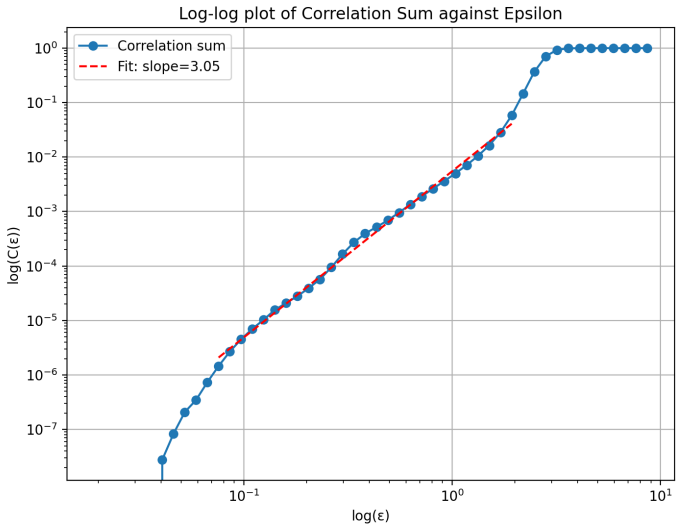

Figure 9: Log-log plot of $C(\varepsilon)$ against ε with a straight line fit for $n = 59$

Figure 9 indicates that the system is chaotic for $n = 59$; it is a necessary condition for chaos that the correlation dimension is greater than 2. Although this is not a sufficient condition, we see that the gradient value of 3.05 is significantly larger than 2.

In order to confirm this let us create a phase plot to observe how to system behaves in phase space. However we have 59 components to our system - we will plot a 3D phase plot using only the first 3 variables; we increase the final time to $tf = 1000000$ to illustrate this:

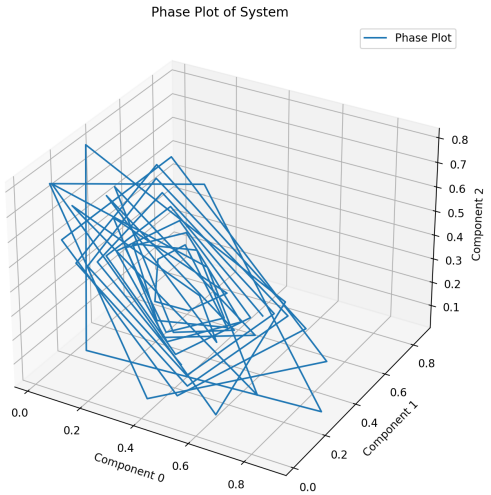

Figure 10: Phase plot of the first 3 components of the system for $n = 59$

This phase plot shows that the first 3 components of the system do not repeat their behaviour, indicating chaos. This is sufficient for chaos; if a subset of the components are chaotic then the system

as a whole is chaotic.

Together this evidence strongly suggests that the system is chaotic for $n = 59$.

n=20:

We use the parameters $Nt = 20000$ and $tf = 10000$; the contour plot is shown below:

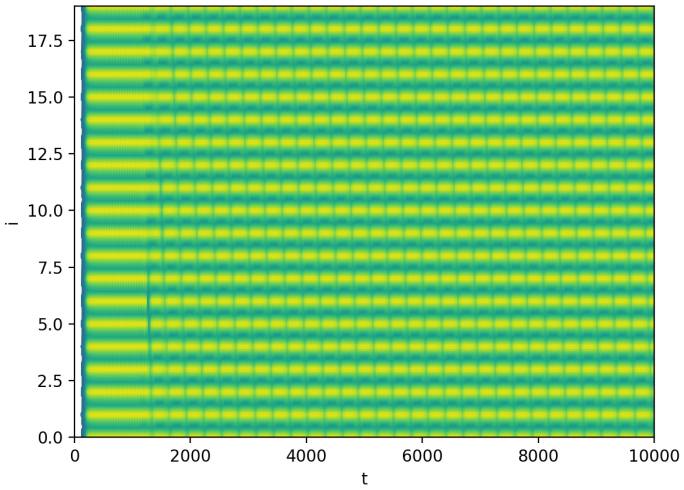

Figure 11: Contour plot for system with parameters $n = 20$, $Nt = 20000$ and $tf = 10000$

Again we can see the effect of the initial condition from $t = 0$ to around $t = 2000$; we remove data preceding $t = 4000$ to be sure we discard the transient (transient_cutoff $= 0.4$). Note that the components seem to be synchronised with a phase difference from this image; when drawing vertical lines we can see that the behaviour of each component is repeat with a slight time delay.

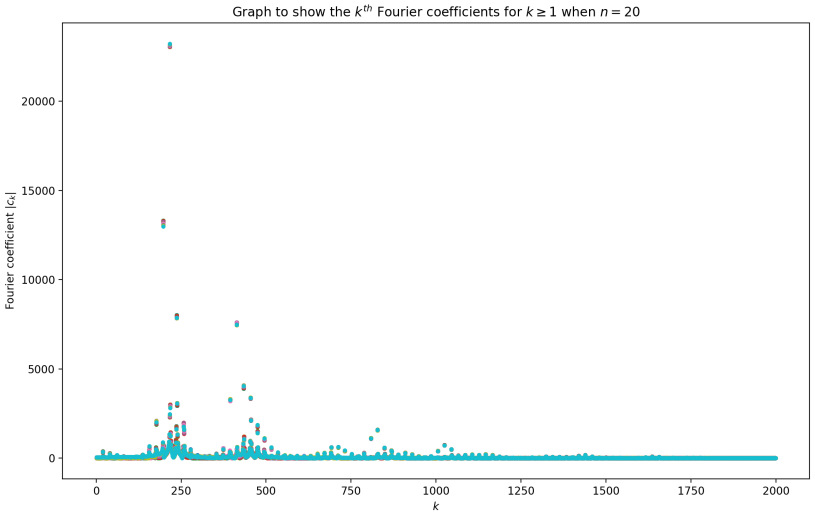

Figure 12: Plot of the first 2000 Fourier coefficients for the system when $n = 20$ excluding c_0

The figure above shows that the Fourier coefficients behave slightly more unpredictably than $n = 9$, but the spectrum is much less wide than compared to $n = 59$. We observe the pattern that peaks of decaying magnitude do occur as k increases; this gives a slight indication of no chaos.

However we notice that all the components of the system have Fourier coefficients peaking at the same values of k. This, in conjunction with the synchronisation argument from the contour plot, indicates

that we still have sinusoidal components, although with different frequencies (unlike $n = 9$). This gives a stronger indication that the system does not have chaotic dynamics.

Again we can plot the correlation sums in order to estimate the correlation dimension.

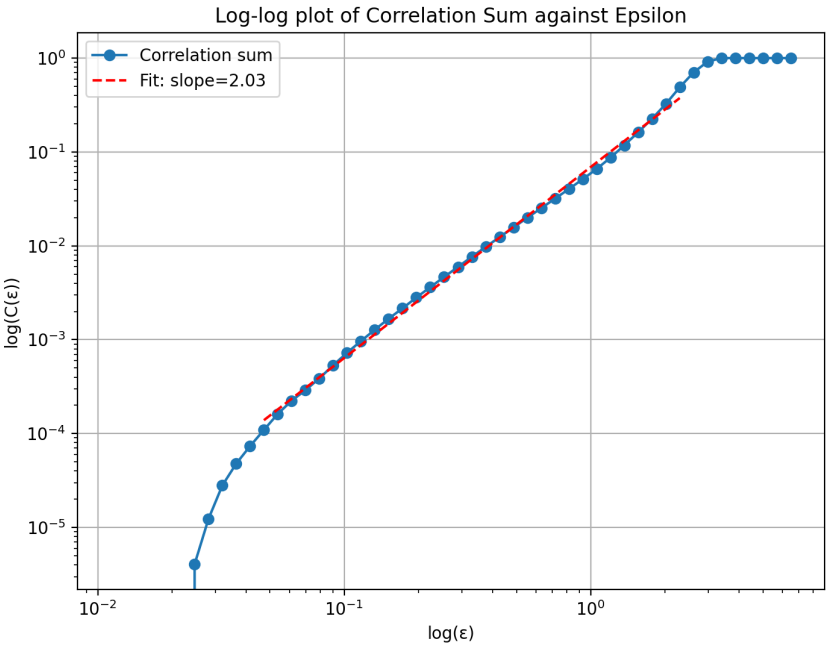

Figure 13: Log-log plot of $C(\varepsilon)$ against ε with a straight line fit for $n = 20$

Computationally we estimate that the correlation dimension is 2.03. This is very close to our value of 2 in our necessary condition and hence it is difficult to judge whether the system exhibits chaos or not. We plot a graph of successive maxima to investigate further:

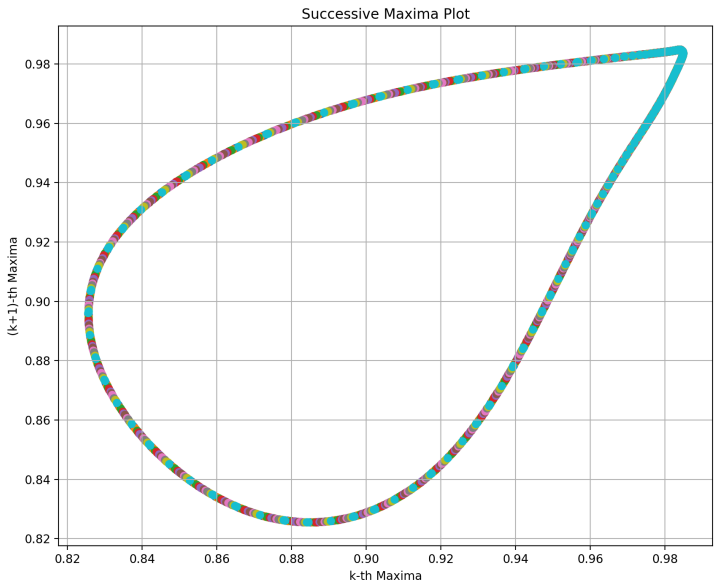

Figure 14: Plot of successive maxima for $n = 20$

Usually we would expect a unimodal/parabolic shape for a chaotic system - instead we have a deformed circular shape. Although we cannot exactly state that we do not have chaos since we do not have a unimodal shape, we have highlighted an area for further analysis which could help solidify any conclusions.

The case $n = 20$ is therefore more subtle than the other 2. We conclude, since the Fourier coefficients have a predictable pattern, and the contour plot seems to indicate synchronisation of components, that the system for $n = 20$ is unlikely to be chaotic.

Part 2

1.

The function part2q1 computes the first and second derivatives in a similar way to dualfd1, but more efficiently. As before, we will solve the system $\mathbf{Af'} = \mathbf{b}$ where $\mathbf{A}$ is a septadiagonal/banded matrix; we will take advantage of this sparse format using scipy.

We generate the matrix $\mathbf{A}$ with the same logic as dualfd1. We start by defining entries using the boundary equations before defining the remaining entries from the equations for the unknowns $u'_{i,j}$ and $u''_{i,j}$. However our method is much more efficient since we only iterate over the seven non-empty diagonals rather than every single row of the matrix. Therefore we only iterate 7 times rather than $2n$ times. We also use vectorised operations in numpy by slicing the arrays rather than editing element-wise.

We also generate $\mathbf{b}$ with the same logic as dualfd1. However, dualfd1 only works for the specific case $m = 1$; to extend this we generate an m x $2n$ matrix where the ith row is the corresponding vector $\mathbf{b}_i$. This is more efficient than generating separate successive vectors $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_m$.

We can then use .data from scipy.sparse to convert our matrix $\mathbf{A}$ into banded/matrix diagonal ordered form. This is much more efficient than using np.toarray; .data takes advantage of the sparsely populated matrix to convert it in $O(1)$ time (we discuss cost in q2).

We have now set up our system so that we can apply solve_banded from scipy to solve the systems each time. We populate the two-dimensional arrays df and d2f accordingly, which are our derivatives.

The function test_part2q1 tests the mechanism of part2q1 using the function

$$f(x, y) = \sin(\pi x) \cdot \cos(\pi y).$$

2.

Here we will compare the cost and accuracy of the methods part2q1 and fd2 using square matrices of dimension n x n (we assume m is comparable to n). We time how long it takes for the two methods to compute the derivatives of the function used in test_part2q1. To do this, we consider matrices of increasing dimension, run both methods on each matrix 10 times and find the average time taken. We then plot a graph of time taken against matrix dimension n as shown below in Figure 11. We discard data at the horizontal boundary.

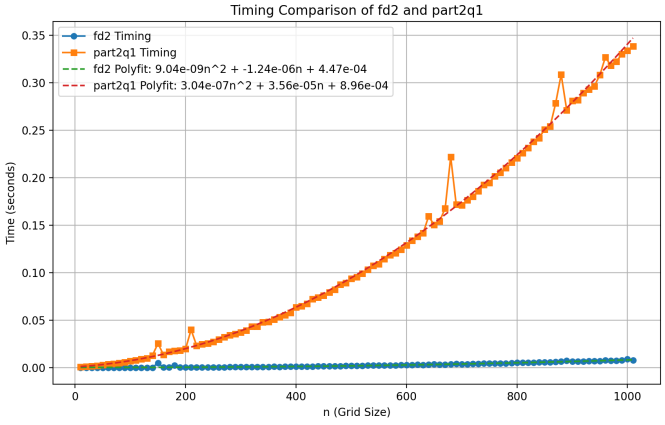

Figure 15: Graph to show how long it takes for methods part2q1 and fd2 to execute depending on matrix dimension n

Figure 11 illustrates the difference in the time taken for the two methods. Note that we have also done a polynomial fit to the scatter graphs in order to estimate the time complexity of the two functions; since they are both quadratic we conjecture that both methods are of complexity $\mathcal{O}(n^2)$ or more generally $\mathcal{O}(mn)$ for non-square matrices.

We can investigate this by analysing the structure of both methods.

In part2q1 the generation of the matrix $\mathbf{A}$ has complexity $\mathcal{O}(n)$: setting entries of the numpy arrays (maind ud1 ud2 ud3 ld1 ld2 and ld3) using slicing has linear time complexity in the number of elements copied (they have length at most n), and we only have a constant number (7) of these arrays to generate.

Turning $\mathbf{A}$ into matrix diagonal/banded form has complexity $\mathcal{O}(1)$ as described in the scipy documentation. This is because the scipy.sparse is designed to deal with sparsely populated matrices efficiently.

Generating the matrix representing $\mathbf{b}$ has complexity $\mathcal{O}(mn)$. We fill the entries of the matrix row by row; this is exactly the same as dualfd1. Since we have m rows and n elements in each row, the complexity is $\mathcal{O}(mn)$.

In order to solve the system, we call the function solve_banded m times. This makes use of a specifically designed LU decomposition algorithm for banded matrices and has complexity $\mathcal{O}(n \cdot l^2)$ where l is the bandwidth of the banded matrix ab: 7. Since this is a constant/independent of n calling solve_banded here is of complexity $\mathcal{O}(n)$, and since we call it m times this stage has complexity $\mathcal{O}(mn)$.

Therefore the entire method part2q1 has complexity $\mathcal{O}(mn)$.

In fd2 an explicit finite difference method is implemented. Again the code is of complexity $\mathcal{O}(mn)$ from setting the interior of the first and second derivatives. Again since numpy slicing has linear time complexity in the number of elements copied, when setting df[:, 1:-1] and d2f[:, 1:-1], we look at every row and (in total) every column of the matrix; as such we have complexity $\mathcal{O}(mn)$.

Despite both being of the same time complexity (asymptotically as n and m grow large), we observe from Figure 11 that fd2 takes less time than part2q1. Visually we can see fd2's datapoints are consistently below those of part2q1, and the coefficient of n^2 for fd2's polyfit curve, 9.04×10^{-9}, is less than that of part2q1's, 3.04×10^{-7}. This is to be expected: fd2 is an explicit method whereas part2q1 is an implicit method. Implicit methods require the solution to a system of equations (which is made more efficient using solve_banded in this case) whereas explicit methods have direct algebraic expressions for the derivatives, and do not require a solution to such a system.

We now analyse the accuracy of the two methods. Here we will consider only the first derivative and focus on the portion of the domain away from the horizontal boundaries. To do this we will calculate the first derivative of a function using fd2, part2q1, and simulate the results using the hard-coded exact first derivative, $\frac{df}{dx}(x, y) = \pi \cos(\pi x) \cos(\pi y)$. Using numpy we can generate 2d arrays which represent the differences between the derivatives calculated using the 2 methods with the analytic derivative, and then take an average of all values. We generate a log-log plot as shown below:

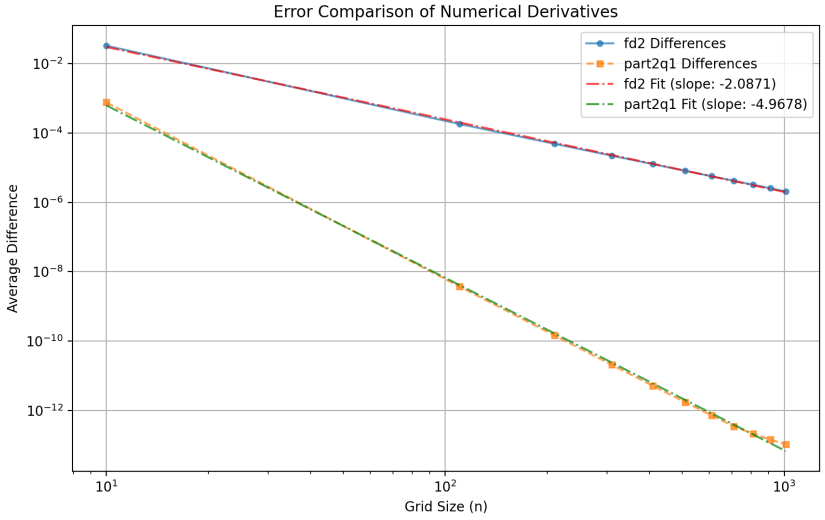

Figure 16: Log-log plot to show how accurate the methods fd2 and part2q1 are as n increases

Having fit a polynomial to both methods, we see the curves are linear and hence establish that both methods exhibit exponential decay; if we define the average error to be E, we have $\log(E) = k \cdot \log(n) + C$ where k is the gradients of each graph and C is the y-intercept. We can rearrange this to get $E = \frac{\exp(C)}{n^{-k}}$.

We will now justify the error behaviour for fd2. If we take the gradient of the line to be -2, we have $E_{fd2} = \frac{\exp(C)}{n^2}$.

From the code for fd2 we can see that it is implementing the following scheme:

$$u'_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2h}$$
$$u''_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

for all $j \in 1, ..., n-2$ i.e. in the interior. We also have the four equations at the boundary. For $j = 0$ we have

$$u'_{i,j} = \frac{-3u_{i,j} + 4u_{i,j+1} - u_{i,j+2}}{2h}$$
$$u''_{i,j} = \frac{2u_{i,j} - 5u_{i,j+1} + 4u_{i,j+2} - u_{i,j+3}}{h^2}$$

and for $j = n-1$ we have

$$u'_{i,j} = \frac{3u_{i,j} - 4u_{i,j-1} + u_{i,j-2}}{2h}$$
$$u'_{i,j} = \frac{2u_{i,j} - 5u_{i,j-1} + 4u_{i,j-2} - u_{i,j-3}}{h^2}$$

where $h = \Delta x$. I have laid this out in a very similar way to how part2q1 was defined.

We now use Taylor's theorem to explain this result. We will do a univariate Taylor expansion in x since y is unchanged by the method.

We know already from lectures that the first equation comes from $\frac{u_{i,j+1} - u_{i,j-1}}{2h} = u'_{i,j} + \mathcal{O}(h^2)$.

Starting with the RHS of the second equation, we have

$$\begin{aligned}
\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} &= \frac{u_{i,j} + u'_{i,j}h + \frac{1}{2}u''_{i,j}h^2 + \frac{1}{6}u'''_{i,j}h^3 + \mathcal{O}(h^4)}{h^2} + \frac{-2u_{i,j}}{h^2} \\
&+ \frac{u_{i,j} + u'_{i,j}(-h) + \frac{1}{2}u''_{i,j}(-h)^2 + \frac{1}{6}u'''_{i,j}(-h)^3 + \mathcal{O}(h^4)}{h^2} \\
&= \frac{u_{i,j} - 2u_{i,j} + u_{i,j}}{h^2} + \frac{u'_{i,j}h + u'_{i,j}(-h)}{h^2} + \frac{\frac{1}{2}u''_{i,j}h^2 + \frac{1}{2}u''_{i,j}(-h)^2}{h^2} \\
&+ \frac{\frac{1}{6}u'''_{i,j}h^3 + \frac{1}{6}u'''_{i,j}(-h)^3}{(-h)^2} + \frac{\mathcal{O}(h^4)}{h^2} \\
&= 0 + 0 + \frac{u''_{i,j}h^2}{h^2} + 0 + \frac{\mathcal{O}(h^4)}{h^2} \\
&= u''_{i,j} + \mathcal{O}(h^2).
\end{aligned}$$

and similarly the RHSs of the other 4 equations all fall out to be the LHS $+\mathcal{O}(h^2)$. So the truncation error is $\mathcal{O}(h^2)$.

[This bit I am not sure about and is speculative] Usually we would argue that the truncation error and global error are related by a factor of h. However in this case we know that the majority of the error comes from the horizontal boundaries and as such the global error will also be $\mathcal{O}(h^2)$.

Note that we have inverse proportion between Δx and n since $\Delta x = \frac{1}{n-1}$. Therefore since the method is $\mathcal{O}(h^2)$ we expect that the relationship between E_{fd2} and n would be inverse square, and hence we have verified our computational findings, namely the straight line behaviour of the curve for fd2 with gradient equal to -2.

The argument for part2q1 is exactly the same as fd2's (using Taylor's theorem to expand). This should justify our gradient value of approximately -5. (I haven't done this since it would be very very long Taylor expansions!)

Looking back at Figure 12, we observe that the error for fd2 is less than that of part2q1's. This is to be expected since part2q1 is an implicit method whereas fd2 is an explicit method.

Therefore we see that, loosely, if we require a faster method we prefer fd2, and if we require a more accurate method we prefer part2q1. To make this more concrete let us plot a graph of time taken to run against accuracy for the two methods.

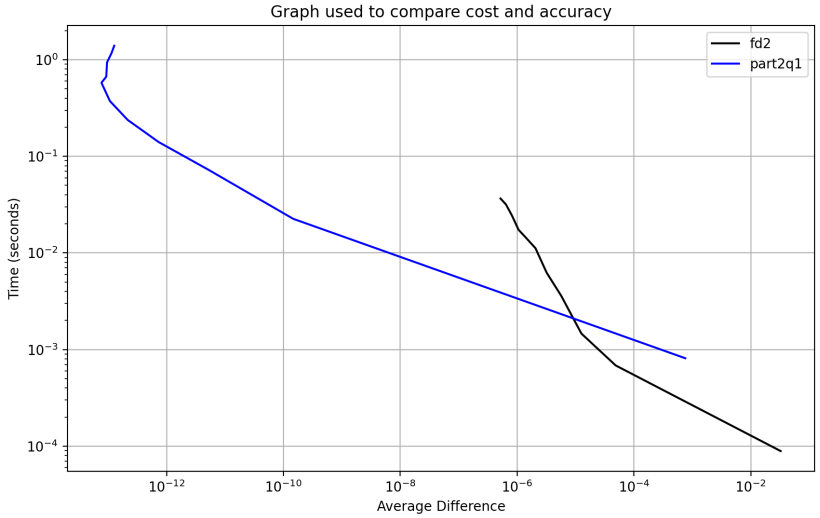

Figure 17: Graph to illustrate the trade-off between cost and accuracy for the 2 methods

This figure illustrates how to choose a method given a desired level of accuracy. In order to compare the 2 methods, consider vertical lines on the graph; for smaller values of accuracy it is faster to use fd2, whereas for larger values of accuracy fd2 would require much longer to compute the derivatives. We can see that for (approximately) an average accuracy of 10^{-5} the two methods require a similar amount of time to run; for greater accuracy we should use part2q1 and vice versa.
